INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Contextual Typesetting of Mathematical Symbols Taking Care of Optical Scaling

Jacques André & Irène Vatton

N° 1972

23 juin 1993

PROGRAMME 3

Intelligence artificielle,
systèmes cognitifs
et interaction homme-machine

Rapport de recherche

1993

ISSN 0249-6399

Contextual Typesetting of Mathematical Symbols Taking Care of Optical Scaling

Jacques André * & Irène Vatton **

Programme 3 — Intelligence artificielle, systèmes cognitifs
et interaction homme-machine
Projet Opéra

Rapport de recherche n°1972 — 23 juin 1993 — 24 pages

Abstract: Typesetting of mathematical formulae has conflicting requirements: on the one hand, optical scaling is a need for large symbols; in the other hand, large symbols are made of composite items that are neither easily nor nicely put together.

In this paper it is shown that such large symbols should be computed at print time so that they reach the quality of metal typesetting. An implementation of such dynamic fonts is in progress in the Grif editor.

Key-words: Mathematical symbols, dynamic font, optical scaling, Grif.

(Résumé : tsvp)

This study is partly financed by the EEC/Comett II project no 90/3697/Cb (Didot: Digitizing and Design of Types).

* Irisa/Inria-Rennes, Campus de Beaulieu, 35042 Rennes cedex – *jandre@irisa.fr*
** CNRS/Inria-Rhônes-Alpes, 2 rue de Vignate, F–38610 Gières, France – *vatton@imag.fr*

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Composition des symboles mathématiques avec ajustement optique et tenant compte du contexte

Résumé : La composition des formules mathématiques pose encore quelques problèmes : d'une part, il faut que les symboles suivent les règles d'ajustement optique ; d'autre part, pour cela, on compose ces grands symboles à l'aide de morceaux plus petits, mais ce faisant on perd la qualité désirée.

Nous montrons ici qu'il est possible de calculer la taille et la forme des grands symboles de façon dynamique, c'est-à-dire lors de l'impression, et d'obtenir ainsi la qualité du temps du plomb.

L'implémentation d'une telle fonte mathématique et son utilisation depuis l'éditeur structuré Grif est en cours.

Mots-clé : Symboles mathématiques, typographie, ajustement optique, fontes dynamiques, Grif,

Contextual Typesetting of Mathematical Symbols
Taking Care of Optical Scaling

Formatting of mathematical formulae is a hard task. Significant progress had been made by the end of the 70s when the tree structure of formulae was used in batch systems such as EQN [1] and TEX [2]. At the beginning of the 80s, systems like Titus [3] allowed interactive formatting of formulae. Today, almost all of the commercial products offer facilities to handle high quality mathematical formulae.

In this paper it is shown that problems still exist: Optical scaling is a must for large symbols: to do so, they are made of smaller items. However, today fonts[1] dedicated to mathematics allow only discrete patterns. A solution is then proposed: it is based on the PostScript font machinery that allows the drawing of a character on the fly and to extend its shape to the correct size according to its context. Finally, an implementation of these principles into the Grif editor is described.

1 Fonts for mathematical formulae

1.1 The needs

Mathematical formulae are made of letters from different families ("a", "a", "α", "$\mathcal{A}$", "$\aleph$", etc.) and of a "menagerie" (as Lamport says [5]) of mathematical symbol ("$\times$", "$\sqcap$", "$\otimes$", "∞", "$\sqsubset$", "$\Leftrightarrow$", etc.) at different point sizes (if, in a^i, a is at 12 point size then i is at 10 point size). Furthermore, miscellaneous rules (e.g. for fractions, determinants, etc.) are needed, as well as a set of variable-sized symbols (integral sign, summation sign, parentheses, horizontal or vertical braces, brackets, arrows, radical sign, etc.).

These variable-sized symbols have to follow the so called *optical scaling* rules: if a 50 point size integral sign was magnified through some affine transformation to 100 point size, this magnification would apply as well to its thickness: the stem would be excessively bold (figure 2). Metal character collections offered large sets of w×h integrals, parentheses, braces, etc. For a given width w measured in points, there was a set of symbols of different heights h. The figure 1 show braces (here numbered from

[1] Here *font* is used with the same meaning as used in Page Description Languages: a font is a set of (algorithms to describe) characters of one familly (e.g. *Helvetica-Narrow-BoldOblique*) that may be rendered in any size, any direction and with any colour.

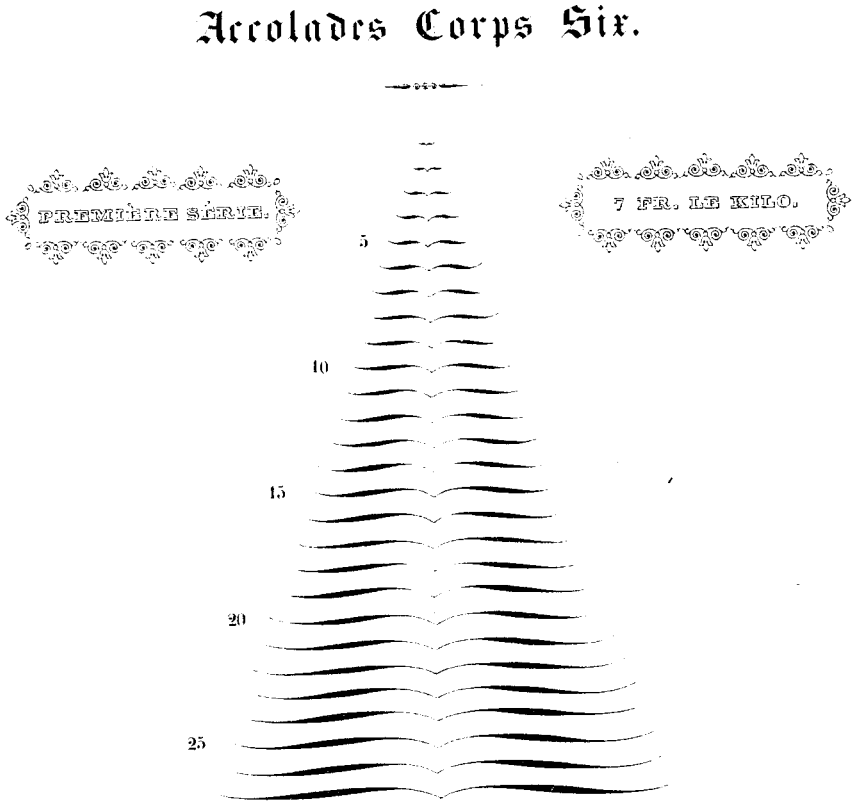

Figure 1: Metal founders offered large sets of variable-sized symbols, at various thickness, and for each, at different sizes. Here, 6 point braces from the *Fonderie Générale* (courtesy *Musée de l'imprimerie, Lyon*).

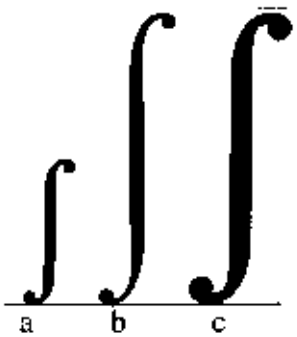

Figure 2: Left: a) " text" integral sign at point size; b) "display" integral sign at 50 point size; c) the same "text" integral as a), magnified about 2 times: note how the thickness changes. (*Lucida* fonts). NOTE: due to copyright problems, the Lucida font can not be included into this PostScript file for distribution through ftp. So, the corresponding characters have been replaced by a bitmap representation of them.

1 to 27) having the same 6 point width. These widths did not vary much (from 3 to 7 or 8 points). However today, bolder symbols are required: if one wants to draw a mathematical formula such as "(a_i)" to be printed on a slide for an overhead projector, one needs to use a large point size, e.g. 60 points, for the "a" and parentheses must have the same boldness (see figure 11 right). This means that, for each given point size, there is a set of variable-sized signs, whose height is adaptable to any factor and whose boldness is (almost) constant.

1.2 Computerized mathematical fonts

Although there are thousands of fonts available on laser printers or phototypesetters, only few of them are concerned with mathematics. Today, the main ones are:

- *cmexnn* (math extension) fonts created by METAFONT for TEX ([4], appendix F);
- *Symbol* font , one of the four fonts that have always been installated on any PostScript system (the three other ones are *Times*, *Courier* and *Helvetica*); refer to [6], appendix E.11;
- The *Lucida* familly that offers both a *LucidaMath-Symbol* almost equivalent to Adobe's *Symbol* and a *LucidaMath-Extension* where one can find symbols that belong to TEX's *cmex* and some other ones; refer to [7].
- Two new versions of *Times* (by Spivak and by Jungers) that offers TEX's math extension as *Lucida* does.

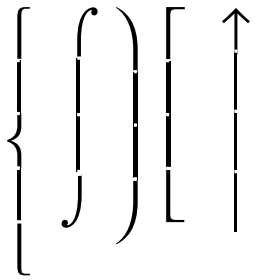

Figure 3: Variable-sized symbols are generally drawn as composite symbols. The spaces between items are only present, here, to indicate the miscellaneous parts of symbols. (*Symbol* font).

As shown in the previous section, two main problems arise with mathematical fonts. Here is the way computerized fonts solve them:

Character set All of these fonts offer a large set of special characters, although always too small. (Another problem with mathematics is that mathematicians are used to inventing new symbols according to their own needs. In that way, any system must be open.). There is no major problem to get bigger character sets. One way is, like *Lucida*, to offer two or more fonts (with the meaning of footnote 1) like Adobe *Expert* fonts do. This is only a problem of software engineering or rather of coding standard (Unicode, a 16 bits encoding scheme, will allow many more symbols; however it will be space consuming).

Variable-sized symbols While mathematics oriented hot metal fonts contained a large set of symbols of any size in any point size, computerized fonts replace variable-sized symbols by a discrete set of composite symbols. For example, the ends of an integral sign are made of the upper spur and of the lower spur while the stem is assembled with a set of spare bars. See figure 3.

1.3 Remaining problems

If formatters use these fonts as they should (by using the complete character set, correct metric, etc.), rather good quality formulae can be drawn. However some problems remain.

1. The construction of composite symbols can be easily done only with truly horizontal or vertical stems (figure 3). However, good mathematics require slanted integral signs. *Symbol* does not offer such slanted integral while TEX's cmex and *Lucida* offer only a limited set of (2 or 3) sizes. This is true as well for other symbols such as the radical sign, big parentheses, horizontal braces, etc.

$$\overbrace{x} = \overbrace{a+b+c+d} \quad \Rightarrow \quad \overbrace{x} = \overbrace{a+b+c+d}$$
$$\widehat{x} = \widehat{a+b+c+d} \quad \Rightarrow \quad \widehat{x} = a + \widehat{b+c} + d$$

Figure 4: LaTeX produces unexpected results with unextensible symbols such as "wide hat".

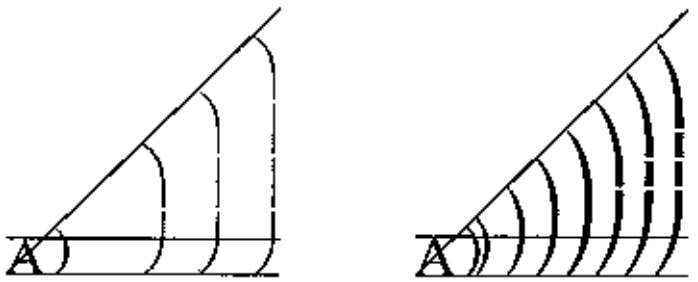

Figure 5: Gaps in variable-sized symbols ar more important in *Symbol* (left) than in *Lucida* (right).
The body size is indicated by the "A" cap height. See figure 2 about the use of bitmaps here.

2. Extending horizontal or vertical strokes does not apply to characters that are not made of single strokes. For example mathematical fonts give the wide hat[2] only a limited set of sizes and no general extension at all. Unexpected (i.e. wrong) results may occur as in figure 4.

3. There is a gap between some character sizes. *Symbol*, see figure 5 left, offers a left parenthesis "(" with a height[3] equal to 0.864em and offers three symbols to compose big right parentheses. The upper part "⎛", the lower part "⎝" and the middle part "⎜". The upper part and the lower part have the same height: 1.220 em. Then, the minimum height[4] for a big parenthesis is 2.440 em. In this font the cap height is 0.673 em. While the parenthesis is approximatively 30% higher than the caps, the minimum big parenthesis is about 330% higher than caps. With such a metric, it is not possible to get parentheses at the right size for expressions as frequent as (a_i). Formatters using *Symbol* have to put extra blanks above and under the a_i or to draw bad parentheses like "⎛⎝".

4. Both *cmex* and *Lucida* fill this gap by offering intermediate medium-size parentheses. See figure 5 right. This solution gives relatively good results according to the respective sizes of these parentheses and of the cap height.

   Nevertheless, perfection is not reached. Let us consider the following LaTeX instruction:

```
\[\left(A^{A^B}\right)+
  \left(A^{A^{A^B}}\right)+
  \left(A^{A^{A^{A^B}}}\right)
\]
```

   that gives the following displayed formula:

$$\left(A^{A^B}\right)+\left(A^{A^{A^B}}\right)+\left(A^{A^{A^{A^B}}}\right)$$

   The first pair of parentheses is set with "big parentheses". The second one with "big big parentheses". The third one too, although the inner box is bigger than in the previous expression. That is why the "B" goes out of the enclosing parentheses of the last expression. Finally, no one parthensis has the right size: good typography would require that each parenthese of this line has the same size, a bit larger than the height of the righter pair. However, this size is not at TeX's disposal. Obviously, this is a matter of hair splitting. However, if things can be upgraded, why not to do so?

[2] It seems that this symbol is more used in France that in English speaking countries.

[3] We call height of a character the difference $ury - lly$, where ury and lly are respectively the ordinates of the upper right corner and of the lower left corner of its bounding box (see [6], 5.4); it is given in terms of em, i.e. relative to the current point size.

[4] Actually, this height may be reduced by 20% by using overlaping of the straight parts of these items.

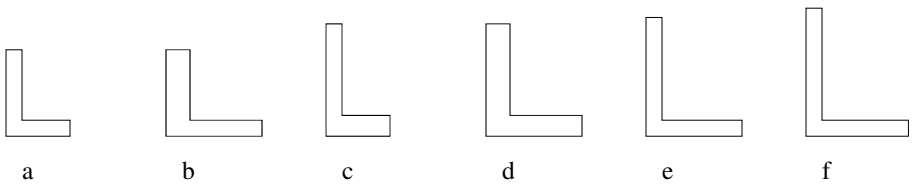

Figure 6: Deforming a "L" with scaling (b to d) and after not-affine transformations (e, f)

The length of the horizontal rule of a radical sign generally is adapted to the specific length of the expression under it. This rule is drawn "on the fly".

If this was possible for any variable-sized symbol, most of the problems quoted in this section would be solved. As it is shown in the next section, this is possible by using the new font machinery available with Page Description Languages such as PostScript[5].

2 Context dependant fonts and PostScript

2.1 PostScript font machinery

Let us consider the following description of the outline of an "L" in a type 3 PostScript font and let us name *Myfont* this font and a call:

```
...

/L{% algorithm to describe ``L''
     0  270 moveto
     0    0 lineto
   200    0 lineto
   200   50 lineto
    50   50 lineto
    50  270 lineto
  closepath stroke  } def

...

/Myfont findfont 150 scalefont setfont
... (L) show              % figure a
```

[5]We assume the reader familiar with PostScript's fonts. If not, he should consider reading [6], [8] and [9].

The problem of the variable-sized symbols is: “how to get the figure 6.e instead of figure 6.a, id est how to increase the size of the stems without increasing their thickness”. The instructions

```
... 1.5 1   scale (L) show   % figure b
... 1   1.3 scale (L) show   % figure c
... 1.5 1.3 scale (L) show   % figure d
```

would respectively give the figures 6.b to 6.d. This is not the expected solution: the boldness of the stems is increased (respectively the vertical bar, the horizontal bar and both are thicker).

The only one way to get the correct shape is to use analytical variables as follows:

```
/movetoy {dy add moveto} def           % x y+dy moveto
/linetoy {dy add lineto} def           % x y+dy lineto
/linetox {exch dx add exch lineto} def % x+dx y lineto

...

/L{% algorithm to analytically describe ``L''
     0  270 movetoy
     0    0 lineto
   200    0 linetox
   200   50 linetox
    50   50 lineto
    50  270 linetoy
  closepath stroke } def
```

and to run the following calling instructions:

```
/dx 100 def % 200x1.5-200
/dy  90 def % 270x1.3-270
(L) show    %  figure e
```

Such analytical computing of characters outline is possible with METAFONT. It is used, for example, to compute arabic or khmer ligatures [10] [11] or even random fonts [12]. However, METAFONT computes the shapes before bitmaps are loaded: analytical variables are not re-evaluated every time a character is used.

Even here, if the following instructions are run:

```
/dx 120 def
/dy 130 def
(L) show % expected figure f
```

the “L” would have exactly the same size as in figure e. Indeed, PostScript font machinery uses a cache memory: when *(L) show* is called the first time, the bitmap of “L” is

Figure 7: This *Punk* poster has been printed by using a single call *(EP - EP - EP - EP) show* – After [16].

computed then saved into a cache memory. At the second call, this cached bitmap is reused, whatever the new values of *dx* and *dy* are. (This is equivalent to a "by value" passing of parameters in programming languages).

However, this caching mechanism may be disabled: the procedure *setcharwidth* with its appropriate parameters[6] has to be used instead of *setcachedevice* from the `BuildChar` procedure. Parameters are then passed "by name" and the figure f is really printed when no cache is used.

In section 4 this mechanism is compared with other ones such as METAFONT or as Adobe *Master fonts*.

2.2 Drawing characters on the fly

Although this mechanism has been exhibited some years ago [13], it does not seem that its power has been significantly understood. Indeed, drawing a character on the fly allows to take care of its context (see [14] for a survey on these contextual fonts). Different contexts may be involved:

1. No context: i.e. purely random. A typical example is the *Scrabble* font where each piece randomly stirs as if they were an earthquake [15]. In [16], the Knuth's *Punk* font [12] has been reprogrammed in such a way that each instantiation of any letter gives a new shape (see figure 7) and, furthermore, hazard is introduced into the curve definitions to give the impression of inkspreading. Other fonts now use these possibilities to simulate antique fonts or to create degraded letters [17], [18]. Commercial products, like *PhotoShop*, allow such deformations; however, they are not dynamic: every time degraded letters are reused (for example from *Illustrator*), the same distorsions are produced. Truly degraded letters require computation at any letter occurrence, like in figure 8.

2. The context of a character may be its neighbours. Ligatures may be automatically defined, like in [19]. Ends of lines or of words are special cases of neighbouring: a contextual font may decide to change the shape of a, say, "s" at the end of a word; or to modify the bearing of a character at the (right or left) end of a line [20].

[6] Because *setcharwidth* does not ask for any caching mechanisme, the bounding box coordinates have not to be passed to.

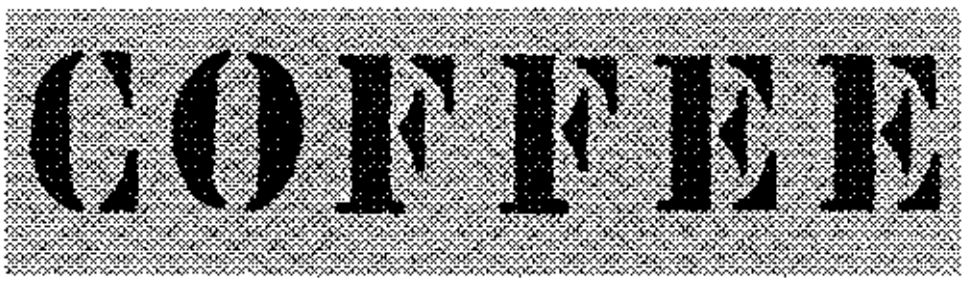

Figure 8: *Pochoir* simulates inkspreading caused by stencil on sackcloth. It recomputes each character degradation at each occurrence (look how each "F" and each "E" are different) – bitmaped fonts (see figure 2).

3. More generally, any typographical requirement such as kerning on the fly, optical scaling, etc. may be considered as a contextual problem. Even, researches are done to get automatic kerning [25].
4. Other contexts may be used, such as phonetic [21] or even stack-mechanims [22].
5. While in the examples 2 to 4, this could be decided by an intelligent editor and by using large sets of characters, more interesting are the fonts where you need continuous variations, like for arabic characters [23] and for . . . mathematics.

Math-Fly is a font we are designing to handle dynamic mathematical symbols. It uses *movetox*, *movetoy*, *linetox*, . . . operators as described in section 2.1. Various classes of problems have to be solved.

2.3 Getting outlines

In this first prototype, we are working with public domain fonts and, when needed, products like *Ikarus, Fontographer*, etc. are used from hand drawings. In any case, we get outline descriptions with numerical values for each outline point[7]. Examples shown here are inspired by *Symbol*.

2.4 Making analytical computation

The main problem is, having outlines described with numerical coordinates, to decide which ones have to be transformed into analytical variables and which values are to be assigned to them.

2.4.1 Linear symbols

[7]We call " outline points" the ends of lines and curves as well as Bézier control points.

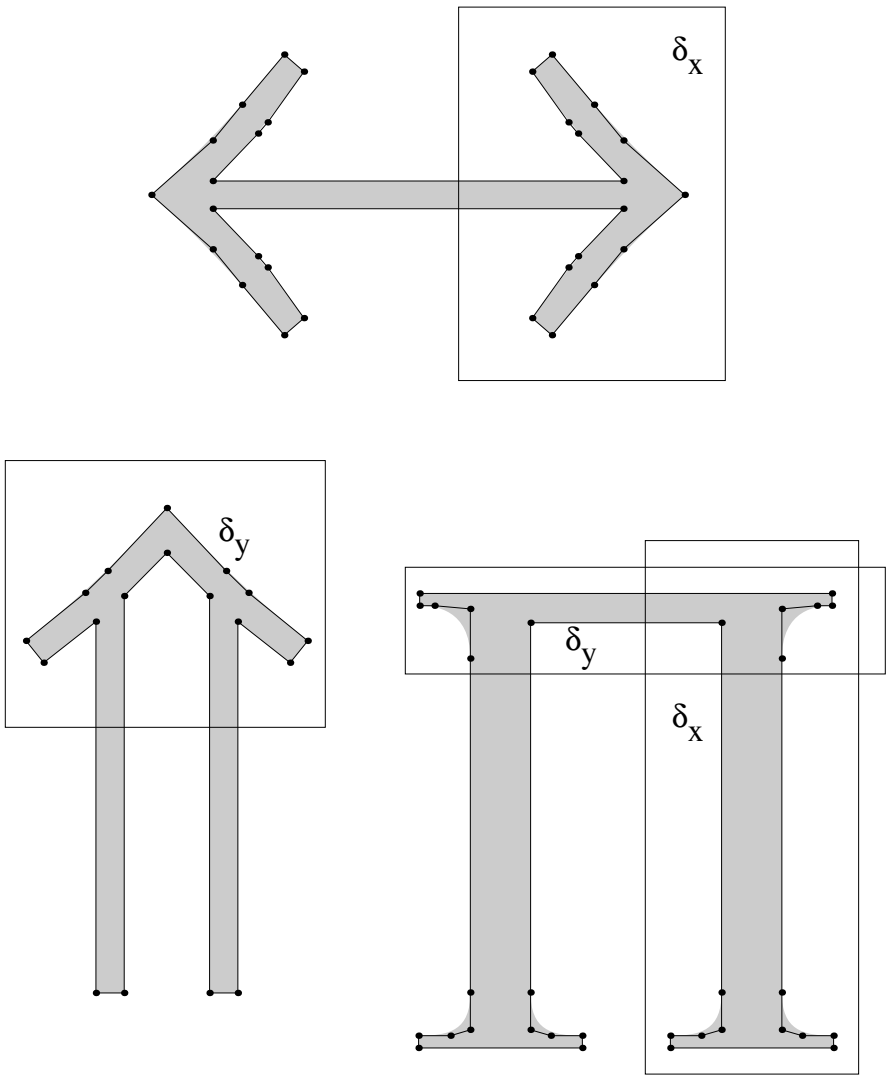

Figure 9: Abscisses of outline points are incremented by δx if they are in the corresponding box; ordinates of outline points are incremented by δy if they are in the corresponding box. Both coordinates are incremented if outline points belong to the two boxes.

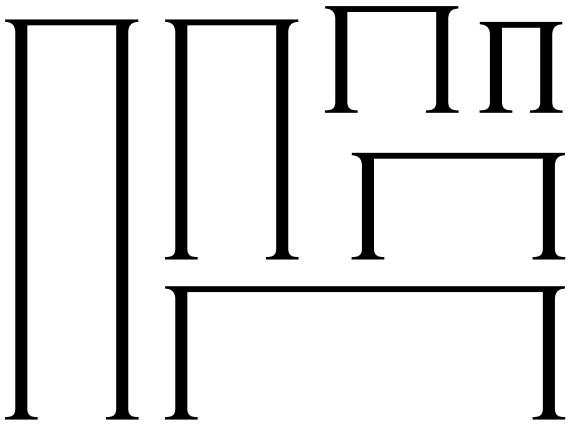

Figure 10: A *Math-Fly* symbol may have its horizontal or vertical straight lines extended while keeping the same thickness. Upper right corner: the regular $\prod$ – see also figure 9

Most of the symbols are made of linear strokes, with outline points only at the ends (junctions, serifs, etc.). Extension of such symbols requires only to globally translate these ends. Horizontal symbols (such as $\Longleftrightarrow$ or $\longmapsto$) abscisses have to be incremented by δx (this value has to be computed by the formatter according to the "content" of this symbol). Vertical symbols (such as $[$, $|$ or $\Uparrow$) ordinates have to be incremented by δy. Some symbols, such as $\sum$, $\prod$ or $\surd$ may support both translations δx and δy. See figures 9 and 10.

2.4.2 Non-linear symbols

The same rule could be applied as well to parentheses, braces, etc. when they are made of linear segments (like in figures 3 and 5). However, if they are curved, a perpendicular deformation is needed as well (see figure 11.right). The same occurs with linear symbols that are not in one direction only, such as hats (figure 11.left).

2.4.3 Optical scaling of large symbols

Outline descriptions of particular symbols, such as integral signs and braces, intensively require Bézier curves. The goal of our font is to increase the size (e.g. the height of an integral sign) without modifying the stem thickness.

Linear scaling can not be used (see figure 2) and optical scaling has to be used instead. However, nonlinear scaling methods (and even principles) have remained largely unexplored. We may only quote recent theses by Bridget Lynn Johnson [24] and by Claude Betrisey [25]. Johnson proposed a nonlinear scaling model, used it to generate letters and compared them with handcut fonts. Betrisey, in the other hand is,

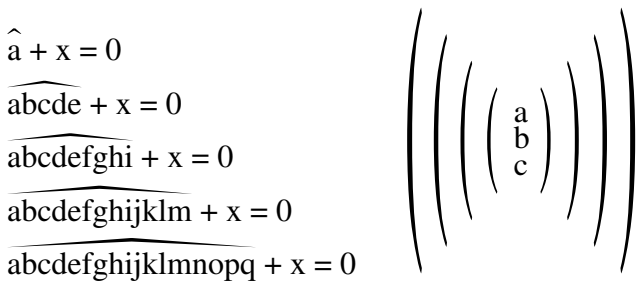

Figure 11: *Math-Fly* transforms non-linear symbols such as hats or parentheses in one direction with a slight perpendicular deformation

more concerned with spacing between letters; however his model should be useful for further developments.

As opposed to regular letters (where optical scaling concerns mainly small body sizes), optical scaling concerns large mathematical symbols which can be more than 4 or 5 centimètres high: in that case, measurements are easily done without much precision errors.

We are studying[8] the way large metal symbols are designed according to their size. The method is to start from old mathematical symbol collections, such as braces in figure 1. Different variable sized braces are scanned and their bitmaps are vectorized (we used Agfa *PressView*). The figure 12-left shows three such braces (magnified 3 times more on x than on y). Irregularities on the curves are probably due to the gouge. Hairlines at the end have been dropped because there was too much noise on them. The figure 12-right exhibits the upper part of these three braces (magnified at the same size[9]) with the Bézier points on it. In this example, the following can be seen: Bézier control points of the bowl are on a linear set (and the ratio A_5A_4/A_5A_2 is approximatively the same as (50-40)/(50-20), where 50, 40 and 20 are the point size of the curves where the points A_5, A_4 and A_2 respectively are. This is less true on the ends of the braces, which may be caused by very small defects at punch time.

Other experiments show that finally linear functions may describe the set of these Bézier control points.

So, we have used the following method

[8]The first results, even if significaint, have to been confirmed with further important set of measures on various fonts, either from printed material, from punches or, better, from "smoked proofs".

[9]If we were using the same scale factor, large braces are less curved than small ones: outlines are overwritten and figure would be less ligible.

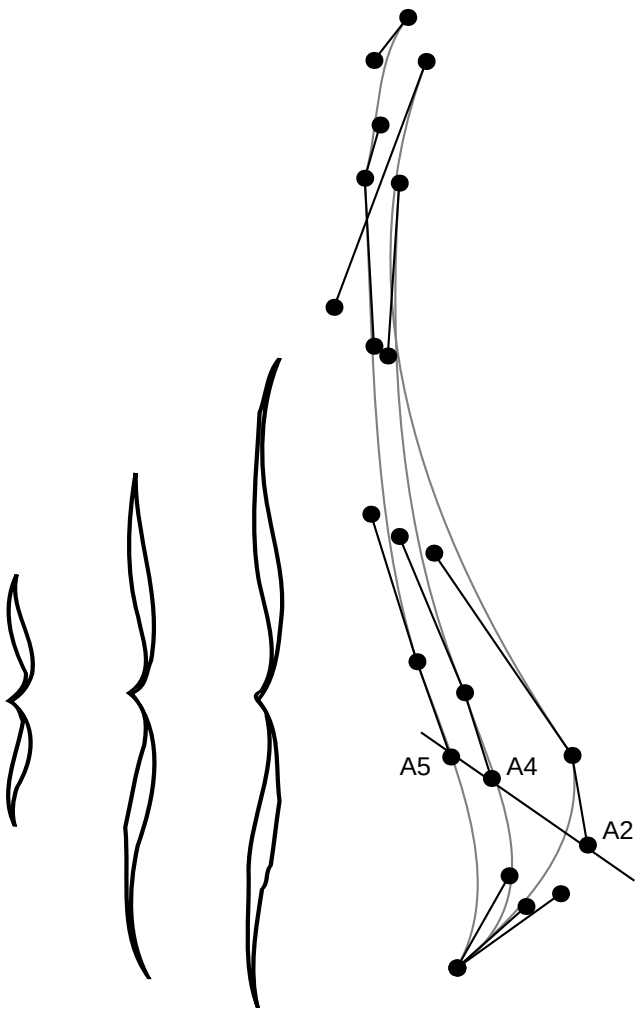

Figure 12: Study of the optical scaling deformation of hot metal braces: left: three braces at the same point size but with different heights; right: their control points are linear.

1. Draw a small integral,
2. Draw a large integral with the same weight as 1.
3. Use linear interpolation between the corresponding outline points. Note that these lines may be parallel or not depending on the Bézier curves (in other words, symbols in the whole do not follow linear scaling).

2.5 Implementation problems

Math-Fly is a true "type 3" PostScript font. However, such fonts are defined as read-only dictionnaries: passing parameters (such as δx and δy), use of variables (to keep intermediate results out of the stack) and of operators (such as *movetox, movetoy*) require the use of a dictionnary out of the font dictionnary.

Furthermore, the standard font metric file has to be modified to give the formatters information on the way symbols are extended.

3 Use of Math-fly from Grif

Due to its possibility to receive parameters, a font like *Math-Fly* could not be fully used by editors such as TEX, Interleaf [30] or Framemaker [28] without modifications of the way they call formulae symbols. Due to our involvement into the Grif project, this editor was a good place to check this font.

3.1 Grif overview

Grif is an interactive system for editing and formatting complex documents [26] [27] where documents are represented by their logical structure rather than by their graphical aspect. In this respect it is comparable to syntax-driven editors used for editing programs. It is based on SGML [29] Document Type Definitions (DTD), which sepcify the logical organization of the document to be processed. More precisely, a DTD specifies the types of the elements constituting a document and of the relationships these elements can have with each other. For example an Article is defined as having a title, one or more authors, an abstract, a body and a bibliography; the body contains a sequence of sections and each section has a heading followed by a variable number of paragraphs, and so on. Grif uses the SGML language to define many DTDs, one for each class of document.

Using such a model, Grif builds a document with a *logical structure* conforming to the DTD. It lets the user enter the *content* of the document (essentially character strings constituting the atomic elements of the structure) and assists him by automatically creating some parts of the document according to the DTD. Grif offers only those elements that may be created at the current position in the document. It computes all numbers (chapter numbers, section numbers, formula numbers...) and updates references when

referenced elements are moved or changed. It permits selection and moving across the document following its logical structure.

All editing commands are performed through a formatted picture of the document displayed on the screen, as in WYSIWYG systems. When modifying the structure or the content of a document the user acts directly on this formatted picture and sees the results of the commands immediately thereon.

As well as the logical structure, the visual aspect of documents (on the screen or on a sheet of paper) is specified on a generic basis. When defining a new DTD, the user gives *presentation rules* for each type of element defined in the generic logical structure, and Grif uses these rules for building the picture of a document. Thus, the user who edits a document has only to enter its logical structure as well as its content, and the system automatically generates its picture.

Grif handles objects of the various types which appear in documents in exactly the same way, whether they be tables, mathematical formulae drawings or pieces of program; in fact all those types of objects which are logically structured and can be described by a DTD. There is no specialized editor for these objects, but a unique environment which handles in a homogeneous way the whole document and all the structured objects it contains.

3.2 Abstract Picture

A specific language, called P language, is used to describe presentation rules for each type of element defined in the DTD. Interpretation of these presentation rules is based on the concept of *abstract pictures* [31]. When Grif has a document to print or to display in a window, it first builds an abstract picture which is a high-level description of that document image. This description is device-independent and allows the editor to update the image dynamically in a simple way. In a second step, it translates this abstract picture into the real image which is displayed on the screen or printed on paper sheets. The abstract picture is the support of interaction between the application and the user.

An abstract picture is a hierarchy of *abstract boxes*, the concept of box being the rectangle which delimits a document element as defined in TEX [4]. In an abstract picture, boxes are organized as a tree, where each node is a box which encloses all its children. These boxes are termed abstract because all their attributes are defined in a relative way. Let us examine a simple example like the following centered formula:

$$m = \sum_{k=min(1,i)}^{n} \sin^2 x_k \qquad (1)$$

The abstract picture describing this formula 1 is shown in Fig. 13 while boxes represented by this abstract picture are shown in Fig. 14

As opposed to PostScript [6], this kind of picture description does not locate document elements in a virtual space, but relative to one another. The tree structure gives a first approximation of relative positions only giving the enclosures amongst boxes.

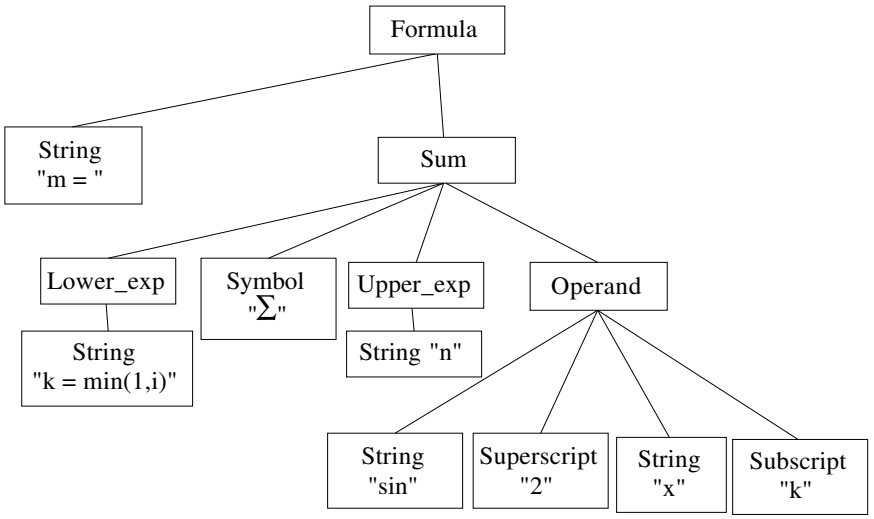

Figure 13: Abstract picture corresponding to the formula 1

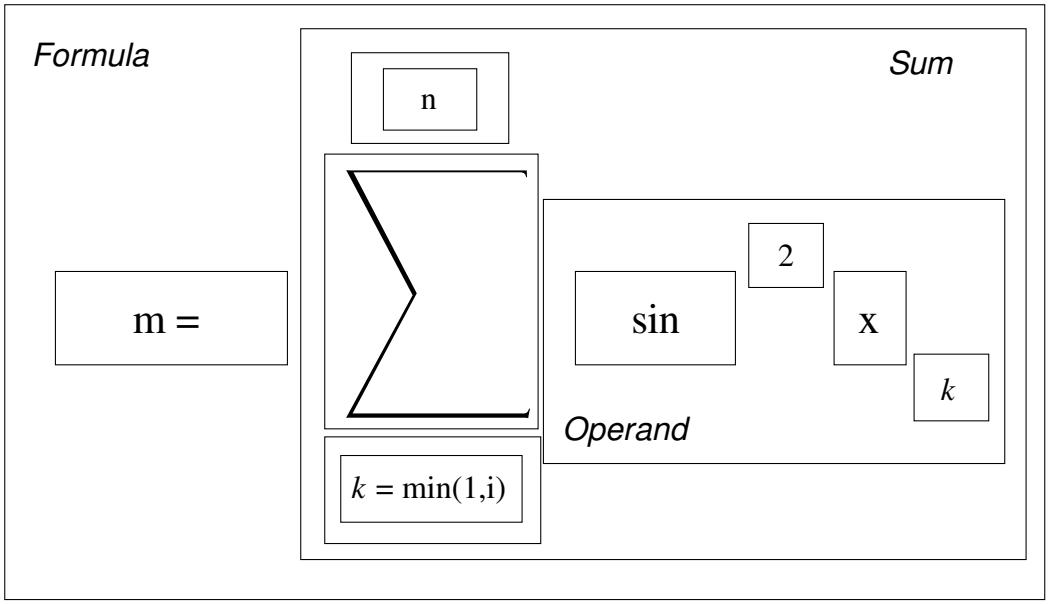

Figure 14: Boxes corresponding to the formula 1

Each node is decorated with position attributes and dimension attributes which express geometric constraints between boxes. These attributes result from constraints expressed in P language.

Therefore each box is located relative to its enclosing box or to one of its sibling boxes, with a fixed distance between two parallel edges or axes of the two boxes. In each direction the position can be defined with reference to a different box. In our example, some position attributes are:

```
Sum:        HorizontalPosition:
                  Left = Previous String.Right;
            VerticalPosition:
                  BaseLine = Previous String.BaseLine;
Symbol:     HorizontalPosition:
                  HorizontalCenter = Lower_exp.HorizontalCenter;
            VerticalPosition:
                  Bottom = Lower_exp.Top;
Upper_exp:  HorizontalPosition:
                  HorizontalCenter = Lower_exp.HorizontalCenter;
            VerticalPosition:
                  Bottom = Symbol.Top;
Operand:    HorizontalPosition:
                  Left = Symbol.Right;
            VerticalPosition:
                  BaseLine = Symbol.BaseLine;
....
```

Positions may use the four edges of a box (Top, Bottom, Left, Right), its center and its base line. The base line of a box may be defined relatively to the box itself or to any of its children. In the example, base lines are defined by:

```
String:  BaseLine = Default;
Sum:     BaseLine = Symbol.BaseLine;
Symbol:  BaseLine = Self VerticalCenter + 0.4;
Operand: BaseLine = Child String.BaseLine;
```

The result of the previous set of constraints is that:

- *String* "m =", `Symbol` "$\sum$" and `String` "sin" are displayed on the same baseline, *Symbol* base line being 0.4 below its middle;
- *Upper exp* is displayed centered above the `Symbol` "$\sum$";
- *Symbol* "$\sum$" is displayed centered above the `Lower exp`.

The dimension of a box can also be specified relatively to the dimension of its enclosing box or one of its sibling boxes. The relation then specifies the difference between the dimensions of the boxes, or a ratio between them. A dimension can also be fixed independently of any other box. Examples of dimension attributes are:

$$m = \sum_{k=\min(1,i)}^{n} \sin^2 x_k$$

Figure 15: Formule 1 as seen by *Math-fly* + Grif

```
Symbol:     Width =  Lower_exp.Width;
            Height = Operand.Height * 1.2;
```

This set of constraints express that:

- *Symbol* width depends on the `Lower_exp` width;
- the height of the *Symbol* "$\sum$" is equal to 1.2 times the height of the *Operand*.

The abstract picture permits an incremental display and so ensures high performance for interactive applications such as an editor which frequently modifies some small parts of a picture. This description allows the application to do the minimum of change to the abstract picture.

Tree structure and constraints between boxes offer a powerful mechanism for describing such complex pictures as frequently found in mathematical formulae. The logical organization of mathematical constructions is described using a DTD and their graphical aspect is specified using the P language. After that, the user has just to manipulate mathematical formulae in logical terms; Grif being in charge to maintain their real images.

3.3 Output drawing

P language allows us to express how to calculate the better size of mathematical symbols (more precisely the size fo their enclosing box), but it's also necessary to have a correct drawing of these symbols. Only *Math-fly* is able to exploit this optimal size evaluation.

Grif computes the abstract picture and its geometrical constraints to obtain the correct position and precise size of symbols to print on paper. Above, the dimension of the box symbol is *width=lower_exp.width*. Conventional formatters fill this box with the nearest available symbol in the font, i.e. with a too small $\sum$, and add exaggerated blank side bearings, like in formula 1.

In conjunction with *Math-fly* font metric file, Grif computes the precise extensions δx and δy to be made on the $\sum$ symbol, and passes these values to the font through a dictionary. Formula 1, when handled by Grif, becomes as in figure 15.

4 Comparaison with other mechanisms

Let us compare four ways of font generation, namely PostScript (with and without cache mechanism), METAFONT and Adobe *Master fonts*[32][33]. Other ones exist, like *MultiType* by URW [20], however they resemble one of these.

1. PostScript, with cache mechanism, works at print time. It receives outline descriptions. At the first instantiation of a character, its bitmap is computed (and cached). Any new instantiation causes to recall the cached bitmap. Analytical variables in the outline description are evaluated only once.

2. PostScript, without cache mechanisms, works at print time. It receives outline descriptions. Any character instantiation causes to (re)compute the bitmap. Any analytical variable is evaluated at each ocurrence.

3. METAFONT works before print time. It receives outline and ductus description, mostly in terms of variables (METAFONT is a meta-font processor). It computes bitmaps that are loaded into the printer then used at any character instantiation. Variables are evaluated during the creation of bitmaps, not when they are used.

4. *MultipleMasters* work in two times: first, a *setup* is processed: starting from an analytical "master" description of outlines. A large set of fonts is created. These fonts are PostScript regular fonts (with cache mechanism) and handled as such: analytical variables are not reevaluated at print time. Furthermore, the number of control points can not be changed from one font to another one.

According to the kind of work, any of these methods has its own advantages. We think that for characters having a great number of variations (like arabic extended characters [23]) or mathematical symbols, dynamic fonts are the best ones. In the other hand, computation of large ligature sets, like for Khmer language [11], may take advantage of METAFONT possibilities.

5 Conclusion

Math-Fly is still in progress, as well as its integration into the Grif editor. However the first results are sufficiently promising to carry on with studies on optical scaling of large symbols and on the automatic marking of outline points to be incremented.

References

[1] B.W. Kernighan and L. L. Cherry, "A system for Typesetting Mathematics", *Communications of the ACM*, 18, 151–157, 1975.

[2] D. Knuth, "Tau Epsilon Chi, a system for technical text", *Stanford Computer Science report* number STAN-CS-78-675, september 1978. Now appears as [4].

[3] V. Quint, "Editing Mathematics on the Buroviseur", in N. Naffah (ed.), *Office Information Systems*, North-Holland pub., 1982, 149–159.

[4] D. Knuth, *The TEXbook*, Addison-Wesley: Reading, 1984.

[5] Leslie Lamport, LATEX, *a document preparation system – user's guide & reference manual*, Addison-Wesley, 1986.

[6] Adobe Systems Incorporated, *PostScript Language Reference Manual*, second edition, Addison-Wesley: Reading, 1991.

[7] C. Bigelow and K. Holmes, "The Design of Lucida: an Integrated Family of Types for Electronic Literacy", in J.C. van Vliet (ed.), *Text Processing and Document Manipulation*, Cambridge University Press, 1986.

[8] Adobe Systems Incorporated, *Adobe Type 1 Font Format*, Addison-Wesley: Reading, 1990.

[9] Adobe Systems Incorporated, *PostScript Language Tutorial and Cookbook*, Addison-Wesley: Reading, 1985.

[10] Yannis Haralambous, "Towards the revival of traditional Arabic typography... through TEX", *EuroTEX 92 proceedings*, (Jiři ZLATUŠKA ed.) Prague, Czechoslovakia, 1992, 293–305.

[11] Yannis Haralambous, " Typesetting Khmer ligatures", paper submitted to *EPODD*, june 1993.

[12] Donal Knuth, "A punk meta-font", *TUGboat*, vol. 9, no. 2, August 1988, 152–168.

[13] Jacques André and Bruno Borghi, "Dynamic fonts", in *raster Imaging and Digital Typography* (J. André and R.D. Hersch eds.), Cambridge University Press, 1989, 198–203. See also, *The PostScript Journal*, vol. 2, no. 3, 1989, 6–8.

[14] Jacques André, *Fontes dynamiques*, Mémoire d'habilitation à diriger les recherches, Université de Rennes, july 1993 (to appear).

[15] Jacques André, "The Scrabble font", *The PostScript Journal*, vol. 3, num. 1, 1990, 53–55.

[16] Jacques André et Victor Ostromoukhov, "Punk: de METAFONT à PostScript", *Cahiers GUTenberg*, 4, 1989, 23–28.

[17] Erik van Blokland and Just van Rossum, "Different Approaches to Lively Outlines", *Raster Imaging and Digital Typogaphy II* (R. Morris and J. André eds.), Cambridge University Press, 1991, 28–33.

[18] " Erik van Blokland & Just van Rossum", *Emigre*, number 18, 1991, p. 23–25.

[19] Jacques André et Christian Delorme, "Le Delorme: un caractère modulaire et dépendant du contexte", *Communication et langage*, 86, 1990, 65–76.

[20] Peter Karow, "hz-program, Micro-typography for advanced typesetting", URW, 1993.

[21] " Pierre di Sciullo", *Emigre*, number 18, 1991,6–22.

[22] Michael Cohen, "Blush and Zebrackets: Two Schemes for Typographical Representation of Nested Associativity", *Proceedings IEEE Workshop on Visual Languages*, October 1992, Seattle, Washington, 264–266.

[23] Johny Srouji and Daniel Berry, "Arabic formatting with *ditroff/ffortid*", *EPODD, Electronic Publishing – Origination, Dissemination and Design*, vo.5, issue No. 4, December 1992, 163–208.

[24] Bridget Lynn Johnson, *A Model for Automatic Optical Scaling of Type Designs for Conventional and Digital Technology*, Master of Science, School Printing in the College of Graphic Arts of hte Rochester Institute of Technology, May 1987.

[25] Claude Bétrisey, *Génération automatique de contraintes pour caractères typographiques à l'aide d'un modèle topologique*, PhD Thesis, Lausanne, 28 juin 1993.

[26] V. Quint & I. Vatton, " Grif: an Interactive System for Structured Document Manipulation", *Text Processing and Document Manipulation* (J.C. van Vliet ed.), Cambridge University Press, 1986, 200–213.

[27] R. Furuta, V. Quint, and J. André, " Interactively Editing Structured Documents", *EPODD, Electronic Publishing – Origination, Dissemination and Design*, vol. 1, issue No. 1, April 1988, 19–44.

[28] FrameMaker, *Reference Manual*, Frame Technology Corporation, San Jose(Calif.), May 1990.

[29] Goldfarb, *The SGML Handbook*, Oxford University Press, Oxford, 1990.

[30] R.A. Morris, "Is What You See Enough To Get?", *Protext II*, (J.J.H. Miller ed.), Boole Press, Dublin, 1985, 56–81.

[31] V. Quint and I. Vatton, "An Abstract Model for Interactive Pictures", *Human-computer interaction, Interact'87*, H.-J. Bullinger and B. Shackel eds., North-Holland, September 1987, 643–647.

[32] Adobe, *MultiMasters specifications*, 1992.

[33] Jonathan Seybold, "Adobe's 'Multimaster' Technology: Breakthrough in Type Aesthetics", *The Seybold Report on Desktop Pubishing*, vol.5, Number 7, March 4, 1991, pages 3–7.

Projet OPÉRA
Récentes publications internes IRISA–PI

Les publications suivantes sont disponibles :

- sous forme papier ; s'adresser au *Service documentation, Irisa, campus de Beaulieu, F-35042 Rennes cedex, France ; FAX: (+33) 99 38 38 32* ;
- pour les plus récentes, par ftp anonyme à *ftp.irisa.fr* (131.254.254.2), dans le répertoire */techreports*, sous forme compressée PI-xxx.ps.Z (où xxx=numéro de publication).

The following research reports from Opera team are available

- in paper form, on request to *Service documentation, Irisa, campus de Beaulieu, F-35042 Rennes cedex, France ; FAX: (+33) 99 38 38 32* ;
- by anonymous ftp at *ftp.irisa.fr*, directory : */techreports*, as compressed files *PI-xxx.ps.Z* where xxx= research report number.

609. Hélène RICHY, Patrice FRISON et Éric PICHERAL, *Intégration d'un correcteur typographique dans l'éditeur structuré Grif*, Publication interne Irisa n° 609, 1991.

636. Jacques ANDRÉ et Roger HERSCH, *Un* curriculum *pour la typographie numérique*, Publication interne Irisa n° 636, 1992.

676. Jacques ANDRÉ, *Font metrics*, Publication interne Irisa n° 676, 1992.

677. Hélène RICHY, *Grif et les index électroniques*, Publication interne Irisa n° 677, 36 pages, septembre 1992.

715. Jacques ANDRÉ, Dominique DECOUCHANT Vincent QUINT et Hélène RICHY, *Vers un atelier éditorial pour les documents structurés*, Publication interne Irisa n° 715, 15 pages, mars 1993.

747. Jacques ANDRÉ and Irène VATTON, *Contextual Typesetting of Mathematical Formulae Taking Care of Optical Scaling*, Publication interne Irisa n° 747, 24 pages, juin 1993.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
ISSN 0249-6399